**ReactJs**

# **Create a Simple JSX Element**

React is an Open Source view library created and maintained by Facebook (Meta). It's a great tool to render the User Interface (UI) of modern web applications.

React uses a syntax extension of JavaScript called JSX that allows you to write HTML directly within JavaScript. This has several benefits. It lets you use the full programmatic power of JavaScript within HTML, and helps to keep your code readable. For the most part, JSX is similar to the HTML that you have already learned, however there are a few key differences that will be covered throughout these challenges.

For instance, because JSX is a syntactic extension of JavaScript, you can actually write JavaScript directly within JSX. To do this, you simply include the code you want to be treated as JavaScript within curly braces:

{ 'this is treated as JavaScript code' }.

Keep this in mind, since it's used in several future challenges.

However, because JSX is not valid JavaScript, JSX code must be compiled into JavaScript. The transpiler Babel is a popular tool for this process. For your convenience, it's already added behind the scenes for these challenges. If you happen to write syntactically invalid JSX, you will see the first test in these challenges fail.

It's worth noting that under the hood the challenges are calling

 ReactDOM.render(JSX, document.getElementById('root')).

This function call is what places your JSX into React's own lightweight representation of the DOM. React then uses snapshots of its own DOM to optimize updating only specific parts of the actual DOM. The current code uses JSX to assign a div element to the constant JSX.

const JSX = <div></div>;

*Replace the div with an h1 element and add the text Hello JSX! inside it.*

**const JSX = <h1>Hello JSX!</h1>;**

# **Create a Complex JSX Element**

The last challenge was a simple example of JSX, but JSX can represent more complex HTML as well.

One important thing to know about nested JSX is that it must return a single element.This one parent element would wrap all of the other levels of nested elements.

For instance, several JSX elements written as siblings with no parent wrapper element will not transpile.

Here's an example:

**Valid JSX:**

<div>

<p>Paragraph One</p>

<p>Paragraph Two</p>

<p>Paragraph Three</p>

</div>

**Invalid JSX:**

<p>Paragraph One</p>

<p>Paragraph Two</p>

<p>Paragraph Three</p>

*Define a new constant JSX that renders a div which contains the following elements in order:*

*An h1, a p, and an unordered list that contains three li items. You can include any text you want within each element.*

**Note:** When rendering multiple elements like this, you can wrap them all in parentheses, but it's not strictly required. Also notice this challenge uses a div tag to wrap all the child elements within a single parent element. If you remove the div, the JSX will no longer transpile. Keep this in mind, since it will also apply when you return JSX elements in React components.

**const JSX=**

**<div>**

**<h1>Fruits</h1>**

**<p>List of fruits</p>**

**<ul>**

**<li>Apple</li>**

**<li>Banana</li>**

**<li>Orange</li>**

**</ul>**

**</div>**

# **Add Comments in JSX**

JSX is a syntax that gets compiled into valid JavaScript. Sometimes, for readability, you might need to add comments to your code. Like most programming languages, JSX has its own way to do this.

To put comments inside JSX, you use the syntax {/\* \*/} to wrap around the comment text.

*Add a comment somewhere within the provided div element, without modifying the existing h1 or p elements.*

**const JSX = (**

**<div>**

**{/\* This is a Comment \*/}**

**<h1>This is a block of JSX</h1>**

**<p>Here's a subtitle</p>**

**</div>**

**);**

# **Render HTML Elements to the DOM**

So far, you've learned that JSX is a convenient tool to write readable HTML within JavaScript. With React, we can render this JSX directly to the HTML DOM using React's rendering API known as ReactDOM.

ReactDOM offers a simple method to render React elements to the DOM which looks like this: ReactDOM.render(componentToRender, targetNode), where the first argument is the React element or component that you want to render, and the second argument is the DOM node that you want to render the component to.

As you would expect, ReactDOM.render() must be called after the JSX element declarations, just like how you must declare variables before using them.

*The code editor has a simple JSX component. Use the ReactDOM.render() method to render this component to the page. You can pass defined JSX elements directly in as the first argument and use document.getElementById() to select the DOM node to render them to. There is a div with id='challenge-node' available for you to use. Make sure you don't change the JSX constant.*

**const JSX = (**

**<div>**

**<h1>Hello World</h1>**

**<p>Lets render this to the DOM</p>**

**</div>**

**);**

// Change code below this line

**ReactDOM.render(JSX,document.getElementById(“challenge-node”));**

# **Define an HTML Class in JSX**

Now that you're getting comfortable writing JSX, you may be wondering how it differs from HTML.

So far, it may seem that HTML and JSX are exactly the same.

One key difference in JSX is that you can no longer use the word class to define HTML classes. This is because class is a reserved word in JavaScript. Instead, JSX uses className.

In fact, the naming convention for all HTML attributes and event references in JSX become camelCase. For example, a click event in JSX is onClick, instead of onclick. Likewise, onchange becomes onChange. While this is a subtle difference, it is an important one to keep in mind moving forward.

*Apply a class of myDiv to the div provided in the JSX code.*

**const JSX = (**

**<div className="myDiv">**

**<h1>Add a class to this div</h1>**

**</div>**

**);**

# **Learn About Self-Closing JSX Tags**

So far, you’ve seen how JSX differs from HTML in a key way with the use of className vs. class for defining HTML classes.

Another important way in which JSX differs from HTML is in the idea of the self-closing tag.

In HTML, almost all tags have both an opening and closing tag: <div></div>; the closing tag always has a forward slash before the tag name that you are closing. However, there are special instances in HTML called “self-closing tags”, or tags that don’t require both an opening and closing tag before another tag can start.

For example the line-break tag can be written as <br> or as <br />, but should never be written as <br></br>, since it doesn't contain any content.

In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. The line-break tag, for example, must always be written as <br /> in order to be valid JSX that can be transpiled. A <div>, on the other hand, can be written as <div /> or <div></div>. The difference is that in the first syntax version there is no way to include anything in the <div />. You will see in later challenges that this syntax is useful when rendering React components.

**const JSX = (**

**<div>**

**<h2>Welcome to React!</h2> <br />**

**<p>Be sure to close all tags!</p>**

**<hr />**

**</div>**

**);**

# **Create a Stateless Functional Component**

Components are the core of React. Everything in React is a component and here you will learn how to create one.

There are two ways to create a React component. The first way is to use a JavaScript function. Defining a component in this way creates a stateless functional component. The concept of state in an application will be covered in later challenges. For now, think of a stateless component as one that can receive data and render it, but does not manage or track changes to that data. (We'll cover the second way to create a React component in the next challenge.)

To create a component with a function, you simply write a JavaScript function that returns either JSX or null. One important thing to note is that React requires your function name to begin with a capital letter. Here's an example of a stateless functional component that assigns an HTML class in JSX:

const DemoComponent = function() {

return (

<div className='customClass' />

);

};

After being transpiled, the <div> will have a CSS class of customClass.

Because a JSX component represents HTML, you could put several components together to create a more complex HTML page. This is one of the key advantages of the component architecture React provides. It allows you to compose your UI from many separate, isolated components. This makes it easier to build and maintain complex user interfaces.

The code editor has a function called MyComponent.

*Complete this function so it returns a single div element which contains some string of text.*

**const MyComponent = function() {**

// Change code below this line

**return(**

**<div>SomeText</div>**

**);**

// Change code above this line

**}**

# **Create a React Component**

The other way to define a React component is with the ES6 class syntax. In the following example, Kitten extends React.Component:

class Kitten extends React.Component {

constructor(props) {

super(props);

}

render() {

return (

<h1>Hi</h1>

);

}

}

This creates an ES6 class Kitten which extends the React.Component class. So the Kitten class now has access to many useful React features, such as local state and lifecycle hooks. Don't worry if you aren't familiar with these terms yet, they will be covered in greater detail in later challenges. Also notice the Kitten class has a constructor defined within it that calls super(). It uses super() to call the constructor of the parent class, in this case React.Component. The constructor is a special method used during the initialization of objects that are created with the class keyword. It is best practice to call a component's constructor with super, and pass props to both. This makes sure the component is initialized properly. For now, know that it is standard for this code to be included. Soon you will see other uses for the constructor as well as props.

MyComponent is defined in the code editor using class syntax. Finish writing the render method so it returns a div element that contains an h1 with the text Hello React!.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

// Change code below this line

**return(**

**<div>**

**<h1>Hello React!</h1>**

**</div>**

**)**

// Change code above this line

**}**

**};**

# **Create a Component with Composition**

Now we will look at how we can compose multiple React components together. Imagine you are building an app and have created three components: a Navbar, Dashboard, and Footer.To compose these components together, you could create an App parent component which renders each of these three components as children. To render a component as a child in a React component, you include the component name written as a custom HTML tag in the JSX. For example, in the render method you could write:

return (

<App>

<Navbar />

<Dashboard />

<Footer />

</App>

)

When React encounters a custom HTML tag that references another component (a component name wrapped in < /> like in this example), it renders the markup for that component in the location of the tag. This should illustrate the parent/child relationship between the App component and the Navbar, Dashboard, and Footer.

*In the code editor, there is a simple functional component called ChildComponent and a class component called ParentComponent. Compose the two together by rendering the ChildComponent within the ParentComponent. Make sure to close the ChildComponent tag with a forward slash.*

**const ChildComponent = () => {**

**return (**

**<div>**

**<p>I am the child</p>**

**</div>**

**);**

**};**

**class ParentComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>I am the parent</h1>**

{ /\* Change code below this line \*/ }

**<ChildComponent/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Use React to Render Nested Components**

The last challenge showed a simple way to compose two components, but there are many different ways you can compose components with React.

Component composition is one of React's powerful features. When you work with React, it is important to start thinking about your user interface in terms of components like the App example in the last challenge. You break down your UI into its basic building blocks, and those pieces become the components. This helps to separate the code responsible for the UI from the code responsible for handling your application logic. It can greatly simplify the development and maintenance of complex projects.

*There are two functional components defined in the code editor, called TypesOfFruit and Fruits. Take the TypesOfFruit component and compose it, or nest it, within the Fruits component. Then take the Fruits component and nest it within the TypesOfFood component. The result should be a child component, nested within a parent component, which is nested within a parent component of its own!*

**const TypesOfFruit = () => {**

**return (**

**<div>**

**<h2>Fruits:</h2>**

**<ul>**

**<li>Apples</li>**

**<li>Blueberries</li>**

**<li>Strawberries</li>**

**<li>Bananas</li>**

**</ul>**

**</div>**

**);**

**};**

**const Fruits = () => {**

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<TypesOfFruit />**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**};**

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

{ /\* Change code below this line \*/ }

**<Fruits />**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Compose React Components**

As the challenges continue to use more complex compositions with React components and JSX, there is one important point to note. Rendering ES6 style class components within other components is no different than rendering the simple components you used in the last few challenges. You can render JSX elements, stateless functional components, and ES6 class components within other components.

In the code editor, the TypesOfFood component is already rendering a component called Vegetables. Also, there is the Fruits component from the last challenge.

*Nest two components inside of Fruits — first NonCitrus, and then Citrus. Both of these components are provided for you behind the scenes. Next, nest the Fruits class component into the TypesOfFood component, below the h1 heading element and above Vegetables. The result should be a series of nested components, which uses two different component types.*

**class Fruits extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h2>Fruits:</h2>**

{ /\* Change code below this line \*/ }

**<NonCitrus/>**

**<Citrus/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

{ /\* Change code below this line \*/ }

**<Fruits/>**

{ /\* Change code above this line \*/ }

**<Vegetables />**

**</div>**

**);**

**}**

**};**

# **Render a Class Component to the DOM**

You may remember using the ReactDOM API in an earlier challenge to render JSX elements to the DOM. The process for rendering React components will look very similar. The past few challenges focused on components and composition, so the rendering was done for you behind the scenes. However, none of the React code you write will render to the DOM without making a call to the ReactDOM API.

Here's a refresher on the syntax: ReactDOM.render(componentToRender, targetNode). The first argument is the React component that you want to render. The second argument is the DOM node that you want to render that component within.

React components are passed into ReactDOM.render() a little differently than JSX elements. For JSX elements, you pass in the name of the element that you want to render. However, for React components, you need to use the same syntax as if you were rendering a nested component, for example ReactDOM.render(<ComponentToRender />, targetNode). You use this syntax for both ES6 class components and functional components.

*Both the Fruits and Vegetables components are defined for you behind the scenes. Render both components as children of the TypesOfFood component, then render TypesOfFood to the DOM. There is a div with id='challenge-node' available for you to use.*

**class TypesOfFood extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>Types of Food:</h1>**

{/\* Change code below this line \*/}

**<Fruits />**

**<Vegetables />**

{/\* Change code above this line \*/}

**</div>**

**);**

**}**

**};**

// Change code below this line

**ReactDOM.render(<TypesOfFood/>,document.getElementById("challenge-node"))**

# **Write a React Component from Scratch**

Now that you've learned the basics of JSX and React components, it's time to write a component on your own. React components are the core building blocks of React applications so it's important to become very familiar with writing them. Remember, a typical React component is an ES6 class which extends React.Component. It has a render method that returns HTML (from JSX) or null. This is the basic form of a React component. Once you understand this well, you will be prepared to start building more complex React projects.

*Define a class MyComponent that extends React.Component. Its render method should return a div that contains an h1 tag with the text: My First React Component! in it. Use this text exactly, the case and punctuation matter. Make sure to call the constructor for your component, too.*

*Render this component to the DOM using ReactDOM.render(). There is a div with id='challenge-node' available for you to use.*

**class MyComponent extends React.Component{**

**constructor(props) {**

**super(props);**

**}**

**render(){**

**return(**

**<div>**

**<h1>My First React Component!</h1>**

**</div> )**

**}**

**}**

**ReactDOM.render(<MyComponent/>,document.getElementById("challenge-node"));**

# **Pass Props to a Stateless Functional Component**

The previous challenges covered a lot about creating and composing JSX elements, functional components, and ES6 style class components in React. With this foundation, it's time to look at another feature very common in React: **props**. In React, you can pass props, or properties, to child components. Say you have an App component which renders a child component called Welcome which is a stateless functional component. You can pass Welcome a user property by writing:

<App>

<Welcome user='Mark' />

</App>

You use **custom HTML attributes** created by you and supported by React to be passed to the component. In this case, the created property user is passed to the component Welcome. Since Welcome is a stateless functional component, it has access to this value like so:

const Welcome = (props) => <h1>Hello, {props.user}!</h1>

It is standard to call this value props and when dealing with stateless functional components, you basically consider it as an argument to a function which returns JSX. You can access the value of the argument in the function body. With class components, you will see this is a little different.

*There are Calendar and CurrentDate components in the code editor. When rendering CurrentDate from the Calendar component, pass in a property of date assigned to the current date from JavaScript's Date object. Then access this prop in the CurrentDate component, showing its value within the p tags. Note that for prop values to be evaluated as JavaScript, they must be enclosed in curly brackets, for instance date={Date()}.*

**const CurrentDate = (props) => {**

**return (**

**<div>**

**{ /\* Change code below this line \*/ }**

**<p>The current date is: {props.date}</p>**

**{ /\* Change code above this line \*/ }**

**</div>**

**);**

**};**

**class Calendar extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>What date is it?</h3>**

{ /\* Change code below this line \*/ }

**<CurrentDate date={Date()}/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Pass an Array as Props**

The last challenge demonstrated how to pass information from a parent component to a child component as props or properties. This challenge looks at how arrays can be passed as props. To pass an array to a JSX element, it must be treated as JavaScript and wrapped in curly braces.

<ParentComponent>

<ChildComponent colors={["green", "blue", "red"]} />

</ParentComponent>

The child component then has access to the array property colors. Array methods such as join() can be used when accessing the property. const ChildComponent = (props) => <p>{props.colors.join(', ')}</p> This will join all colors array items into a comma separated string and produce: <p>green, blue, red</p> Later, we will learn about other common methods to render arrays of data in React.

*There are List and ToDo components in the code editor. When rendering each List from the ToDo component, pass in a tasks property assigned to an array of to-do tasks, for example ["walk dog", "workout"]. Then access this tasks array in the List component, showing its value within the p element. Use join(", ") to display the props.tasksarray in the p element as a comma separated list. Today's list should have at least 2 tasks and tomorrow's should have at least 3 tasks.*

**const List = (props) => {**

{ /\* Change code below this line \*/ }

**return <p>{props.tasks.join(', ')}</p>**

{ /\* Change code above this line \*/ }

**};**

**class ToDo extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h1>To Do Lists</h1>**

**<h2>Today</h2>**

{ /\* Change code below this line \*/ }

**<List tasks={["Running","Swimming"]}/>**

**<h2>Tomorrow</h2>**

**<List tasks={["Gaming","Drawing","Singing"]}/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Use Default Props**

React also has an option to set default props. You can assign default props to a component as a property on the component itself and React assigns the default prop if necessary. This allows you to specify what a prop value should be if no value is explicitly provided. For example, if you declare MyComponent.defaultProps = { location: 'San Francisco' }, you have defined a location prop that's set to the string San Francisco, unless you specify otherwise. React assigns default props if props are undefined, but if you pass null as the value for a prop, it will remain null.

*The code editor shows a ShoppingCart component. Define default props on this component which specify a prop items with a value of 0.*

**const ShoppingCart = (props) => {**

**return (**

**<div>**

**<h1>Shopping Cart Component</h1>**

**</div>**

**)**

**};**

// Change code below this line

**ShoppingCart.defaultProps={items:0}**

# **Override Default Props**

The ability to set default props is a useful feature in React. The way to override the default props is to explicitly set the prop values for a component.

The ShoppingCart component now renders a child component Items. This Items component has a default prop quantity set to the integer 0. *Override the default prop by passing in a value of 10 for quantity.*

**Note:** Remember that the syntax to add a prop to a component looks similar to how you add HTML attributes. However, since the value for quantity is an integer, it won't go in quotes but it should be wrapped in curly braces. For example, {100}. This syntax tells JSX to interpret the value within the braces directly as JavaScript.

**const Items = (props) => {**

**return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>**

**}**

**Items.defaultProps = {**

**quantity: 0**

**}**

**class ShoppingCart extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**{ /\* Change code below this line \*/ }**

**return <Items quantity={10}/>**

**{ /\* Change code above this line \*/ }**

**}**

**};**

# **Use PropTypes to Define the Props You Expect**

React provides useful type-checking features to verify that components receive props of the correct type. For example, your application makes an API call to retrieve data that you expect to be in an array, which is then passed to a component as a prop. You can set propTypes on your component to require the data to be of type array. This will throw a useful warning when the data is of any other type.

It's considered a best practice to set propTypes when you know the type of a prop ahead of time. You can define a propTypes property for a component in the same way you defined defaultProps. Doing this will check that props of a given key are present with a given type. Here's an example to require the type function for a prop called handleClick:

MyComponent.propTypes = { handleClick: PropTypes.func.isRequired }

In the example above, the PropTypes.func part checks that handleClick is a function. Adding isRequired tells React that handleClick is a required property for that component. You will see a warning if that prop isn't provided. Also notice that func represents function. Among the seven JavaScript primitive types, function and boolean (written as bool) are the only two that use unusual spelling. In addition to the primitive types, there are other types available. For example, you can check that a prop is a React element. Please refer to all of the options.

optionalArray: PropTypes.array,

optionalBool: PropTypes.bool,

optionalFunc: PropTypes.func,

optionalNumber: PropTypes.number,

optionalObject: PropTypes.object,

optionalString: PropTypes.string,

optionalSymbol: PropTypes.symbol

**Note:** As of React v15.5.0, PropTypes is imported independently from React, like this: import PropTypes from 'prop-types';

*Define propTypes for the Items component to require quantity as a prop and verify that it is of type number.*

**const Items = (props) => {**

**return <h1>Current Quantity of Items in Cart: {props.quantity}</h1>**

**};**

// Change code below this line

**Items.propTypes={quantity:PropTypes.number.isRequired}**

// Change code above this line

**Items.defaultProps = {**

**quantity: 0**

**};**

**class ShoppingCart extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <Items />**

**}**

**};**

# **Access Props Using this.props**

The last several challenges covered the basic ways to pass props to child components. But what if the child component that you're passing a prop to is an ES6 class component, rather than a stateless functional component? The ES6 class component uses a slightly different convention to access props.

Anytime you refer to a class component within itself, you use the this keyword. To access props within a class component, you preface the code that you use to access it with this. For example, if an ES6 class component has a prop called data, you write {this.props.data} in JSX.

*Render an instance of the Welcome component in the parent component App. Here, give Welcome a prop of name and assign it a value of a string. Within the child, Welcome, access the name prop within the strong tags.*

**class App extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<Welcome name={"Prateek"}/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

**class Welcome extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<p>Hello, <strong>{this.props.name}</strong>!</p>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Review Using Props with Stateless Functional Components**

Except for the last challenge, you've been passing props to stateless functional components. These components act like pure functions. They accept props as input and return the same view every time they are passed the same props. You may be wondering what state is, and the next challenge will cover it in more detail. Before that, here's a review of the terminology for components.

A stateless functional component is any function you write which accepts props and returns JSX. A stateless component, on the other hand, is a class that extends React.Component, but does not use internal state (covered in the next challenge). Finally, a stateful component is a class component that does maintain its own internal state. You may see stateful components referred to simply as components or React components.

A common pattern is to try to minimize statefulness and to create stateless functional components wherever possible. This helps contain your state management to a specific area of your application. In turn, this improves development and maintenance of your app by making it easier to follow how changes to state affect its behavior.

The code editor has a CampSite component that renders a Camper component as a child. Define the Camper component and assign it default props of { name: 'CamperBot' }. Inside the Camper component, render any code that you want, but make sure to have one p element that includes only the name value that is passed in as a prop. Finally, define propTypes on the Camper component to require name to be provided as a prop and verify that it is of type string.

**class CampSite extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<Camper/>**

**</div>**

**);**

**}**

**};**

// Change code below this line

**const Camper = props => <p>{props.name}</p>;**

**Camper.defaultProps = {**

**name: "CamperBot"**

**};**

**Camper.propTypes = {**

**name: PropTypes.string.isRequired**

**};**

# **Create a Stateful Component**

One of the most important topics in React is state. State consists of any data your application needs to know about, that can change over time. You want your apps to respond to state changes and present an updated UI when necessary. React offers a nice solution for the state management of modern web applications.

You create state in a React component by declaring a state property on the component class in its constructor. This initializes the component with state when it is created. The state property must be set to a JavaScript object. Declaring it looks like this:

this.state = {

}

*You have access to the state object throughout the life of your component. You can update it, render it in your UI, and pass it as props to child components. The state object can be as complex or as simple as you need it to be. Note that you must create a class component by extending React.Component in order to create state like this.*

*There is a component in the code editor that is trying to render a firstName property from its state. However, there is no state defined. Initialize the component with state in the constructor and assign your name to a property of firstName.*

**class StatefulComponent extends React.Component {**

**constructor(props) {**

**super(props);**

// Only change code below this line

**this.state={**

**firstName:"Prateek"**

**}**

// Only change code above this line

**}**

**render() {**

**return (**

**<div>**

**<h1>{this.state.firstName}</h1>**

**</div>**

**);**

**}**

**};**

# **Render State in the User Interface**

Once you define a component's initial state, you can display any part of it in the UI that is rendered. If a component is stateful, it will always have access to the data in state in its render() method. You can access the data with this.state.

If you want to access a state value within the return of the render method, you have to enclose the value in curly braces.

state is one of the most powerful features of components in React. It allows you to track important data in your app and render a UI in response to changes in this data. If your data changes, your UI will change. React uses what is called a virtual DOM, to keep track of changes behind the scenes. When state data updates, it triggers a re-render of the components using that data - including child components that received the data as a prop. React updates the actual DOM, but only where necessary. This means you don't have to worry about changing the DOM. You simply declare what the UI should look like.

Note that if you make a component stateful, no other components are aware of its state. Its state is completely encapsulated, or local to that component, unless you pass state data to a child component as props. This notion of encapsulated state is very important because it allows you to write certain logic, then have that logic contained and isolated in one place in your code.

*In the code editor, MyComponent is already stateful. Define an h1 tag in the component's render method which renders the value of name from the component's state.*

***Note:****The h1 should only render the value from state and nothing else. In JSX, any code you write with curly braces { } will be treated as JavaScript. So to access the value from state just enclose the reference in curly braces.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'freeCodeCamp'**

**}**

**}**

**render() {**

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<h1>{this.state.name}</h1>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Render State in the User Interface Another Way**

There is another way to access state in a component. In the render() method, before the return statement, you can write JavaScript directly. For example, you could declare functions, access data from state or props, perform computations on this data, and so on. Then, you can assign any data to variables, which you have access to in the return statement.

*In the MyComponent render method, define a const called name and set it equal to the name value in the component's state. Because you can write JavaScript directly in this part of the code, you don't have to enclose this reference in curly braces.*

*Next, in the return statement, render this value in an h1 tag using the variable name. Remember, you need to use the JSX syntax (curly braces for JavaScript) in the return statement*.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'freeCodeCamp'**

**}**

**}**

**render() {**

// Change code below this line

**const name=this.state.name;**

// Change code above this line

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<h1>{name}</h1>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

# **Set State with this.setState**

The previous challenges covered component state and how to initialize state in the constructor. There is also a way to change the component's state. React provides a method for updating component state called setState. You call the setState method within your component class like so: this.setState(), passing in an object with key-value pairs. The keys are your state properties and the values are the updated state data. For instance, if we were storing a username in state and wanted to update it, it would look like this:

this.setState({ username: 'Lewis'});

*React expects you to never modify state directly*, instead always use this.setState() when state changes occur. Also, you should note that React may batch multiple state updates in order to improve performance. What this means is that state updates through the setState method can be asynchronous. There is an alternative syntax for the setState method which provides a way around this problem. This is rarely needed but it's good to keep it in mind

There is a button element in the code editor which has an onClick() handler. This handler is triggered when the button receives a click event in the browser, and runs the handleClick method defined on MyComponent. *Within the handleClick method, update the component state using this.setState(). Set the name property in state to equal the string React Rocks!.*

Click the button and watch the rendered state update. Don't worry if you don't fully understand how the click handler code works at this point. It's covered in upcoming challenges.

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = { name: 'Initial State'};**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick() {**

// Change code below this line

**this.setState({**

**name:'React Rocks!'**

**})**

// Change code above this line

**}**

**render() {**

**return (**

**<div>**

**<button onClick={this.handleClick}>Click Me</button>**

**<h1>{this.state.name}</h1>**

**</div>**

**);**

**}**

**};**

# **Bind 'this' to a Class Method**

In addition to setting and updating state, you can also define methods for your component class. A class method typically needs to use the this keyword so it can access properties on the class (such as state and props) inside the scope of the method. There are a few ways to allow your class methods to access this.

One common way is to explicitly bind this in the constructor so this becomes bound to the class methods when the component is initialized. You may have noticed the last challenge used this.handleClick = this.handleClick.bind(this) for its handleClick method in the constructor. Then, when you call a function like this.setState() within your class method, this refers to the class and will not be undefined.

**Note:** The this keyword is one of the most confusing aspects of JavaScript but it plays an important role in React. Although its behavior here is totally normal, these lessons aren't the place for an in-depth review of this so please refer to other lessons if the above is confusing!

*The code editor has a component with a state that keeps track of the text. It also has a method which allows you to set the text to You clicked!. However, the method doesn't work because it's using the this keyword that is undefined. Fix it by explicitly binding this to the handleClick() method in the component's constructor.Next, add a click handler to the button element in the render method. It should trigger the handleClick() method when the button receives a click event. Remember that the method you pass to the onClick handler needs curly braces because it should be interpreted directly as JavaScript.Once you complete the above steps you should be able to click the button and see You clicked!.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = { text: "Hello" };**

// Change code below this line

**this.handleClick = this.handleClick.bind(this);**

// Change code above this line

**}**

**handleClick() {**

**this.setState({**

**text: "You clicked!"**

**});**

**}**

**render() {**

**return (**

**<div>**

**{ /\* Change code below this line \*/ }**

**<button onClick={this.handleClick}>Click Me</button>**

**{ /\* Change code above this line \*/ }**

**<h1>{this.state.text}</h1>**

**</div>**

**);**

**}**

**};**

# **Use State to Toggle an Element**

Sometimes you might need to know the previous state when updating the state. However, state updates may be asynchronous - this means React may batch multiple setState() calls into a single update. This means you can't rely on the previous value of this.state or this.props when calculating the next value. So, you should not use code like this:

this.setState({

counter: this.state.counter + this.props.increment

});

Instead, you should pass setState a function that allows you to access state and props. Using a function with setState guarantees you are working with the most current values of state and props. This means that the above should be rewritten as:

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

You can also use a form without props if you need only the state:

this.setState(state => ({

counter: state.counter + 1

}));

Note that you have to wrap the object literal in parentheses, otherwise JavaScript thinks it's a block of code.

*MyComponent has a visibility property which is initialized to false. The render method returns one view if the value of visibility is true, and a different view if it is false.*

*Currently, there is no way of updating the visibility property in the component's state. The value should toggle back and forth between true and false. There is a click handler on the button which triggers a class method called toggleVisibility(). Pass a function to setState to define this method so that the state of visibility toggles to the opposite value when the method is called. If visibility is false, the method sets it to true, and vice versa.*

*Finally, click the button to see the conditional rendering of the component based on its state.*

**Hint:** Don't forget to bind the this keyword to the method in the constructor!

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**visibility: false**

**};**

// Change code below this line

**this.toggleVisibility=this.toggleVisibility.bind(this);**

// Change code above this line

**}**

// Change code below this line

**toggleVisibility() {**

**this.setState(state => {**

**if (state.visibility === true) {**

**return { visibility: false };**

**} else {**

**return { visibility: true };**

**}**

**});**

**}**

// Change code above this line

**render() {**

**if (this.state.visibility) {**

**return (**

**<div>**

**<button onClick={this.toggleVisibility}>Click Me</button>**

**<h1>Now you see me!</h1>**

**</div>**

**);**

**} else {**

**return (**

**<div>**

**<button onClick={this.toggleVisibility}>Click Me</button>**

**</div>**

**);**

**}**

**}**

**}**

# **Write a Simple Counter**

You can design a more complex stateful component by combining the concepts covered so far. These include initializing state, writing methods that set state, and assigning click handlers to trigger these methods.

*The Counter component keeps track of a count value in state. There are two buttons which call methods increment() and decrement(). Write these methods so the counter value is incremented or decremented by 1 when the appropriate button is clicked. Also, create a reset() method so when the reset button is clicked, the count is set to 0.*

**Note:** Make sure you don't modify the classNames of the buttons. Also, remember to add the necessary bindings for the newly-created methods in the constructor.

**class Counter extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**count: 0**

**};**

// Change code below this line

**this.increment=this.increment.bind(this);**

**this.decrement=this.decrement.bind(this);**

**this.reset=this.reset.bind(this);**

// Change code above this line

**}**

// Change code below this line

**increment(){**

**this.setState({count:this.state.count+1})**

**}**

**decrement(){**

**this.setState({count:this.state.count-1})**

**}**

**reset(){**

**this.setState({count:0})**

**}**

// Change code above this line

**render() {**

**return (**

**<div>**

**<button className='inc' onClick={this.increment}>Increment!</button>**

**<button className='dec' onClick={this.decrement}>Decrement!</button>**

**<button className='reset' onClick={this.reset}>Reset</button>**

**<h1>Current Count: {this.state.count}</h1>**

**</div>**

**);**

**}**

**};**

# **Create a Controlled Input**

Your application may have more complex interactions between state and the rendered UI. For example, form control elements for text input, such as input and textarea, maintain their own state in the DOM as the user types. With React, you can move this mutable state into a React component's state. The user's input becomes part of the application state, so React controls the value of that input field. Typically, if you have React components with input fields the user can type into, it will be a controlled input form.

The code editor has the skeleton of a component called ControlledInput to create a controlled input element. The component's state is already initialized with an input property that holds an empty string. This value represents the text a user types into the input field.

*First, create a method called handleChange() that has a parameter called event. When the method is called, it receives an event object that contains a string of text from the input element. You can access this string with event.target.value inside the method. Update the input property of the component's state with this new string.*

*In the render method, create the input element above the h4 tag. Add a value attribute which is equal to the input property of the component's state. Then add an onChange() event handler set to the handleChange() method.When you type in the input box, that text is processed by the handleChange() method, set as the input property in the local state, and rendered as the value in the input box on the page. The component state is the single source of truth regarding the input data.*

*Last but not least, don't forget to add the necessary bindings in the constructor*.

**class ControlledInput extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = { input: ' '};**

// Change code below this line

**this.handleChange=this.handleChange.bind(this)**

// Change code above this line

**}**

// Change code below this line

**handleChange=(event)=>{**

**this.setState({input:event.target.value})**

**}**

// Change code above this line

**render() {**

**return (**

**<div>**

{ /\* Change code below this line \*/}

**<input value={this.state.input} onChange={this.handleChange}/>**

{ /\* Change code above this line \*/}

**<h4>Controlled Input:</h4>**

**<p>{this.state.input}</p>**

**</div>**

**); }};**

# **Create a Controlled Form**

The last challenge showed that React can control the internal state for certain elements like input and textarea, which makes them controlled components. This applies to other form elements as well, including the regular HTML form element.

The MyForm component is set up with an empty form with a submit handler. The submit handler will be called when the form is submitted.

We've added a button which submits the form. You can see it has the type set to submit indicating it is the button controlling the form. *Add the input element in the form and set its value and onChange() attributes like the last challenge. You should then complete the handleSubmit method so that it sets the component state property submit to the current input value in the local state.*

***Note:****You also must call event.preventDefault() in the submit handler, to prevent the default form submit behavior which will refresh the web page. For camper convenience, the default behavior has been disabled here to prevent refreshes from resetting challenge code.*

*Finally, create an h1 tag after the form which renders the submit value from the component's state. You can then type in the form and click the button (or press enter), and you should see your input rendered to the page*.

**class MyForm extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: '',**

**submit: ''**

**};**

**this.handleChange = this.handleChange.bind(this);**

**this.handleSubmit = this.handleSubmit.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**input: event.target.value**

**});**

**}**

**handleSubmit(event) {**

// Change code below this line

**event.preventDefault()**

**this.setState({**

**submit: this.state.input**

**})**

// Change code above this line

**}**

**render() {**

**return (**

**<div>**

**<form onSubmit={this.handleSubmit}>**

{/\* Change code below this line \*/}

**<input value={this.state.input} onChange={this.handleChange}/>**

{/\* Change code above this line \*/}

**<button type='submit'>Submit!</button>**

**</form>**

{/\* Change code below this line \*/}

**<h1>{this.state.submit}</h1>**

{/\* Change code above this line \*/}

**</div>**

**);**

**}**

**}**

# **Pass State as Props to Child Components**

You saw a lot of examples that passed props to child JSX elements and child React components in previous challenges. You may be wondering where those props come from. A common pattern is to have a stateful component containing the state important to your app, that then renders child components. You want these components to have access to some pieces of that state, which are passed in as props.

For example, maybe you have an App component that renders a Navbar, among other components. In your App, you have state that contains a lot of user information, but the Navbar only needs access to the user's username so it can display it. You pass that piece of state to the Navbar component as a prop.

This pattern illustrates some important paradigms in React. The first is unidirectional data flow. State flows in one direction down the tree of your application's components, from the stateful parent component to child components. The child components only receive the state data they need. The second is that complex stateful apps can be broken down into just a few, or maybe a single, stateful component. The rest of your components simply receive state from the parent as props, and render a UI from that state. It begins to create a separation where state management is handled in one part of code and UI rendering in another. This principle of separating state logic from UI logic is one of React's key principles. When it's used correctly, it makes the design of complex, stateful applications much easier to manage.

*The MyApp component is stateful and renders a Navbar component as a child. Pass the name property in its state down to the child component, then show the name in the h1 tag that's part of the Navbar render method. name should appear after the text Hello, my name is:.*

**class MyApp extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**name: 'CamperBot'**

**}**

**}**

**render() {**

**return (**

**<div>**

{/\* Change code below this line \*/}

**<Navbar name={this.state.name}/>**

{/\* Change code above this line \*/}

**</div>**

**);**

**}**

**};**

**class Navbar extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

{/\* Change code below this line \*/}

**<h1>Hello, my name is: {this.props.name}</h1>**

{/\* Change code above this line \*/}

**</div>**

**);**

**}**

**};**

# **Pass a Callback as Props**

You can pass state as props to child components, but you're not limited to passing data. You can also pass handler functions or any method that's defined on a React component to a child component. This is how you allow child components to interact with their parent components. You pass methods to a child just like a regular prop. It's assigned a name and you have access to that method name under this.props in the child component.

*There are three components outlined in the code editor. The MyApp component is the parent that will render the GetInput and RenderInput child components. Add the GetInput component to the render method in MyApp, then pass it a prop called input assigned to inputValue from MyApp's state. Also create a prop called handleChange and pass the input handler handleChange to it.*

*Next, add RenderInput to the render method in MyApp, then create a prop called input and pass the inputValue from state to it. Once you are finished you will be able to type in the input field in the GetInput component, which then calls the handler method in its parent via props. This updates the input in the state of the parent, which is passed as props to both children. Observe how the data flows between the components and how the single source of truth remains the state of the parent component. Admittedly, this example is a bit contrived, but should serve to illustrate how data and callbacks can be passed between React components.*

**class MyApp extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**inputValue: ''**

**}**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**inputValue: event.target.value**

**});**

**}**

**render() {**

**return (**

**<div>**

{ /\* Change code below this line \*/ }

**<GetInput input={this.state.inputValue} handleChange={this.handleChange}/>**

**<RenderInput input={this.state.inputValue}/>**

{ /\* Change code above this line \*/ }

**</div>**

**);**

**}**

**};**

**class GetInput extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>Get Input:</h3>**

**<input**

**value={this.props.input}**

**onChange={this.props.handleChange}/>**

**</div>**

**);**

**}**

**};**

**class RenderInput extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<div>**

**<h3>Input Render:</h3>**

**<p>{this.props.input}</p>**

**</div>**

**);**

**}**

**};**

# **Use the Lifecycle Method componentWillMount**

React components have several special methods that provide opportunities to perform actions at specific points in the lifecycle of a component. These are called lifecycle methods, or lifecycle hooks, and allow you to catch components at certain points in time. This can be before they are rendered, before they update, before they receive props, before they unmount, and so on. Here is a list of some of the main lifecycle methods: componentWillMount() componentDidMount() shouldComponentUpdate() componentDidUpdate() componentWillUnmount() The next several lessons will cover some of the basic use cases for these lifecycle methods.

**Note:** The componentWillMount Lifecycle method will be deprecated in a future version of 16.X and removed in version 17.

*The componentWillMount() method is called before the render() method when a component is being mounted to the DOM. Log something to the console within componentWillMount() - you may want to have your browser console open to see the output.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**componentWillMount() {**

// Change code below this line

**console.log("This will run");**

// Change code above this line

**}**

**render() {**

**return <div />**

**}**

**};**

# **Use the Lifecycle Method componentDidMount**

Most web developers, at some point, need to call an API endpoint to retrieve data. If you're working with React, it's important to know where to perform this action.

The best practice with React is to place API calls or any calls to your server in the lifecycle method componentDidMount(). This method is called after a component is mounted to the DOM. Any calls to setState() here will trigger a re-rendering of your component. When you call an API in this method, and set your state with the data that the API returns, it will automatically trigger an update once you receive the data.

*There is a mock API call in componentDidMount(). It sets state after 2.5 seconds to simulate calling a server to retrieve data. This example requests the current total active users for a site. In the render method, render the value of activeUsers in the h1 after the text Active Users:. Watch what happens in the preview, and feel free to change the timeout to see the different effects.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**activeUsers: null**

**};**

**}**

**componentDidMount() {**

**setTimeout(() => {**

**this.setState({**

**activeUsers: 1273**

**});**

**}, 2500);**

**}**

**render() {**

**return (**

**<div>**

{/\* Change code below this line \*/}

**<h1>Active Users: {this.state.activeUsers}</h1>**

{/\* Change code above this line \*/}

**</div>**

**);**

**}**

**}**

# **Add Event Listeners**

The componentDidMount() method is also the best place to attach any event listeners you need to add for specific functionality. React provides a synthetic event system which wraps the native event system present in browsers. This means that the synthetic event system behaves exactly the same regardless of the user's browser - even if the native events may behave differently between different browsers.

You've already been using some of these synthetic event handlers such as onClick(). React's synthetic event system is great to use for most interactions you'll manage on DOM elements. However, if you want to attach an event handler to the document or window objects, you have to do this directly.

*Attach an event listener in the componentDidMount() method for keydown events and have these events trigger the callback handleKeyPress(). You can use document.addEventListener() which takes the event (in quotes) as the first argument and the callback as the second argument.*

*Then, in componentWillUnmount(), remove this same event listener. You can pass the same arguments to document.removeEventListener(). It's good practice to use this lifecycle method to do any clean up on React components before they are unmounted and destroyed. Removing event listeners is an example of one such clean up action.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**message: ''**

**};**

**this.handleEnter = this.handleEnter.bind(this);**

**this.handleKeyPress = this.handleKeyPress.bind(this);**

**}**

// Change code below this line

**componentDidMount() {**

**document.addEventListener("keydown", this.handleKeyPress);**

**}**

**componentWillUnmount() {**

**document.removeEventListener("keydown", this.handleKeyPress);**

**}**

// Change code above this line

**handleEnter() {**

**this.setState((state) => ({**

**message: state.message + 'You pressed the enter key! '**

**}));**

**}**

**handleKeyPress(event) {**

**if (event.keyCode === 13) {**

**this.handleEnter();**

**}**

**}**

**render() {**

**return (**

**<div>**

**<h1>{this.state.message}</h1>**

**</div>**

**);**

**}**

**};**

# **Optimize Re-Renders with shouldComponentUpdate**

So far, if any component receives new state or new props, it re-renders itself and all its children. This is usually okay. But React provides a lifecycle method you can call when child components receive new state or props, and declare specifically if the components should update or not. The method is shouldComponentUpdate(), and it takes nextProps and nextState as parameters.

This method is a useful way to optimize performance. For example, the default behavior is that your component re-renders when it receives new props, even if the props haven't changed. You can use shouldComponentUpdate() to prevent this by comparing the props. The method must return a boolean value that tells React whether or not to update the component. You can compare the current props (this.props) to the next props (nextProps) to determine if you need to update or not, and return true or false accordingly.

*The shouldComponentUpdate() method is added in a component called OnlyEvens. Currently, this method returns true so OnlyEvens re-renders every time it receives new props. Modify the method so OnlyEvens updates only if the value of its new props is even. Click the Add button and watch the order of events in your browser's console as the lifecycle hooks are triggered.*

**class OnlyEvens extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**shouldComponentUpdate(nextProps, nextState) {**

**console.log('Should I update?');**

// Change code below this line

**if (nextProps.value % 2 == 0) {**

**return true;**

**}**

**return false;**

// Change code above this line

**}**

**componentDidUpdate() {**

**console.log('Component re-rendered.');**

**}**

**render() {**

**return <h1>{this.props.value}</h1>;**

**}**

**}**

**class Controller extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**value: 0**

**};**

**this.addValue = this.addValue.bind(this);**

**}**

**addValue() {**

**this.setState(state => ({**

**value: state.value + 1**

**}));**

**}**

**render() {**

**return (**

**<div>**

**<button onClick={this.addValue}>Add</button>**

**<OnlyEvens value={this.state.value} />**

**</div>**

**);**

**}**

**}**

# **Introducing Inline Styles**

There are other complex concepts that add powerful capabilities to your React code. But you may be wondering about the more simple problem of how to style those JSX elements you create in React. You likely know that it won't be exactly the same as working with HTML because of [the way you apply classes to JSX elements](https://www.freecodecamp.org/learn/front-end-development-libraries/react/define-an-html-class-in-jsx).

If you import styles from a stylesheet, it isn't much different at all. You apply a class to your JSX element using the className attribute, and apply styles to the class in your stylesheet. Another option is to apply inline styles, which are very common in ReactJS development.

You apply inline styles to JSX elements similar to how you do it in HTML, but with a few JSX differences. Here's an example of an inline style in HTML:

<div style="color: yellow; font-size: 16px">Mellow Yellow</div>

JSX elements use the style attribute, but because of the way JSX is transpiled, you can't set the value to a string. Instead, you set it equal to a JavaScript object. Here's an example:

<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>

Notice how we camelCase the fontSize property? This is because React will not accept kebab-case keys in the style object. React will apply the correct property name for us in the HTML.

*Add a style attribute to the div in the code editor to give the text a color of red and font size of 72px.*

*Note that you can optionally set the font size to be a number, omitting the units px, or write it as 72px.*

**class Colorful extends React.Component {**

**render() {**

**return (**

**<div style={{color:"red",fontSize:72}}>Big Red</div>**

**);**

**}**

**};**

# **Add Inline Styles in React**

You may have noticed in the last challenge that there were several other syntax differences from HTML inline styles in addition to the style attribute set to a JavaScript object. First, the names of certain CSS style properties use camel case. For example, the last challenge set the size of the font with fontSize instead of font-size. Hyphenated words like font-size are invalid syntax for JavaScript object properties, so React uses camel case. As a rule, any hyphenated style properties are written using camel case in JSX.

All property value length units (like height, width, and fontSize) are assumed to be in px unless otherwise specified. If you want to use em, for example, you wrap the value and the units in quotes, like {fontSize: "4em"}. Other than the length values that default to px, all other property values should be wrapped in quotes.

If you have a large set of styles, you can assign a style object to a constant to keep your code organized.

*Declare your styles constant as a global variable at the top of the file. Initialize styles constant and assign an object with three style properties and their values to it. Give the div a color of purple, a font-size of 40, and a border of 2px solid purple. Then set the style attribute equal to the styles constant.*

**const styles={**

**color:"purple",**

**fontSize:40,**

**border:"2px solid purple"**

**}**

// Change code above this line

**class Colorful extends React.Component {**

**render() {**

// Change code below this line

**return (**

**<div style={styles}>Style Me!</div>**

**);**

// Change code above this line

**}**

**};**

# **Use Advanced JavaScript in React Render Method**

In previous challenges, you learned how to inject JavaScript code into JSX code using curly braces, { }, for tasks like accessing props, passing props, accessing state, inserting comments into your code, and most recently, styling your components. These are all common use cases to put JavaScript in JSX, but they aren't the only way that you can utilize JavaScript code in your React components.

You can also write JavaScript directly in your render methods, before the return statement, ***without*** inserting it inside of curly braces. This is because it is not yet within the JSX code. When you want to use a variable later in the JSX code inside the return statement, you place the variable name inside curly braces.

*In the code provided, the render method has an array that contains 20 phrases to represent the answers found in the classic 1980's Magic Eight Ball toy. The button click event is bound to the ask method, so each time the button is clicked a random number will be generated and stored as the randomIndex in state. On line 52, delete the string change me! and reassign the answer const so your code randomly accesses a different index of the possibleAnswers array each time the component updates. Finally, insert the answer const inside the p tags.*

**const inputStyle = {**

**width: 235,**

**margin: 5**

**};**

**class MagicEightBall extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**userInput: '',**

**randomIndex: ''**

**};**

**this.ask = this.ask.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**ask() {**

**if (this.state.userInput) {**

**this.setState({**

**randomIndex: Math.floor(Math.random() \* 20),**

**userInput: ''**

**});**

**}**

**}**

**handleChange(event) {**

**this.setState({**

**userInput: event.target.value**

**});**

**}**

**render() {**

**const possibleAnswers = [**

**'It is certain',**

**'It is decidedly so',**

**'Without a doubt',**

**'Yes, definitely',**

**'You may rely on it',**

**'As I see it, yes',**

**'Outlook good',**

**'Yes',**

**'Signs point to yes',**

**'Reply hazy try again',**

**'Ask again later',**

**'Better not tell you now',**

**'Cannot predict now',**

**'Concentrate and ask again',**

**"Don't count on it",**

**'My reply is no',**

**'My sources say no',**

**'Most likely',**

**'Outlook not so good',**

**'Very doubtful'**

**];**

**const answer = possibleAnswers[this.state.randomIndex];** // Change this line

**return (**

**<div>**

**<input**

**type='text'**

**value={this.state.userInput}**

**onChange={this.handleChange}**

**style={inputStyle}**

**/>**

**<br />**

**<button onClick={this.ask}>Ask the Magic Eight Ball!</button>**

**<br />**

**<h3>Answer:</h3>**

**<p>**

{/\* Change code below this line \*/}

**{answer}**

{/\* Change code above this line \*/}

**</p>**

**</div>**

**);**

**}**

**}**

# **Render with an If-Else Condition**

Another application of using JavaScript to control your rendered view is to tie the elements that are rendered to a condition. When the condition is true, one view renders. When it's false, it's a different view. You can do this with a standard if/else statement in the render() method of a React component.

*MyComponent contains a boolean in its state which tracks whether you want to display some element in the UI or not. The button toggles the state of this value. Currently, it renders the same UI every time. Rewrite the render() method with an if/else statement so that if display is true, you return the current markup. Otherwise, return the markup without the h1 element.*

***Note:****You must write an if/else to pass the tests. Use of the ternary operator will not pass here.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = { display: true }**

**this.toggleDisplay = this.toggleDisplay.bind(this);**

**}**

**toggleDisplay() {**

**this.setState({**

**display: !this.state.display**

**});**

**}**

**render() {**

// change code below this line

**if (this.state.display) {**

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**<h1>Displayed!</h1>**

**</div> );**

**} else {**

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**</div>);**

**}**

**}**

**};**

# **Use && for a More Concise Conditional**

The if/else statements worked in the last challenge, but there's a more concise way to achieve the same result. Imagine that you are tracking several conditions in a component and you want different elements to render depending on each of these conditions. If you write a lot of else if statements to return slightly different UIs, you may repeat code which leaves room for error. Instead, you can use the && logical operator to perform conditional logic in a more concise way. This is possible because you want to check if a condition is true, and if it is, return some markup. Here's an example:

{condition && <p>markup</p>}

If the condition is true, the markup will be returned. If the condition is false, the operation will immediately return false after evaluating the condition and return nothing. You can include these statements directly in your JSX and string multiple conditions together by writing && after each one. This allows you to handle more complex conditional logic in your render() method without repeating a lot of code.

*Solve the previous example again, so the h1 only renders if display is true, but use the && logical operator instead of an if/else statement.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**display: true**

**}**

**this.toggleDisplay = this.toggleDisplay.bind(this);**

**}**

**toggleDisplay() {**

**this.setState(state => ({**

**display: !state.display**

**}));**

**}**

**render() {**

// Change code below this line

**return (**

**<div>**

**<button onClick={this.toggleDisplay}>Toggle Display</button>**

**{this.state.display && <h1>Displayed!</h1>}**

**</div>**

**);**

**}**

**};**

# **Use a Ternary Expression for Conditional Rendering**

Before moving on to dynamic rendering techniques, there's one last way to use built-in JavaScript conditionals to render what you want: the ternary operator. The ternary operator is often utilized as a shortcut for if/else statements in JavaScript. They're not quite as robust as traditional if/else statements, but they are very popular among React developers. One reason for this is because of how JSX is compiled, if/else statements can't be inserted directly into JSX code. You might have noticed this a couple challenges ago — when an if/else statement was required, it was always outside the return statement. Ternary expressions can be an excellent alternative if you want to implement conditional logic within your JSX. Recall that a ternary operator has three parts, but you can combine several ternary expressions together. Here's the basic syntax:

*The code editor has three constants defined within the CheckUserAge component's render() method. They are called buttonOne, buttonTwo, and buttonThree. Each of these is assigned a simple JSX expression representing a button element. First, initialize the state of CheckUserAge with input and userAge both set to values of an empty string.*

*Once the component is rendering information to the page, users should have a way to interact with it. Within the component's return statement, set up a ternary expression that implements the following logic: when the page first loads, render the submit button, buttonOne, to the page. Then, when a user enters their age and clicks the button, render a different button based on the age. If a user enters a number less than 18, render buttonThree. If a user enters a number greater than or equal to 18, render buttonTwo.*

**const inputStyle = {**

**width: 235,**

**margin: 5**

**};**

**class CheckUserAge extends React.Component {**

**constructor(props) {**

**super(props);**

// Change code below this line

**this.state={**

**input:' ',**

**userAge:' '**

**}**

// Change code above this line

**this.submit = this.submit.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(e) {**

**this.setState({**

**input: e.target.value,**

**userAge: ' '**

**});**

**}**

**submit() {**

**this.setState(state => ({**

**userAge: state.input**

**}));**

**}**

**render() {**

**const buttonOne = <button onClick={this.submit}>Submit</button>;**

**const buttonTwo = <button>You May Enter</button>;**

**const buttonThree = <button>You Shall Not Pass</button>;**

**return (**

**<div>**

**<h3>Enter Your Age to Continue</h3>**

**<input style={inputStyle} type='number' value={this.state.input} onChange={this.handleChange} />**

**<br />**

{/\* Change code below this line \*/}

**{**

**this.state.userAge === ''**

**? buttonOne**

**: this.state.userAge >= 18**

**? buttonTwo**

**: buttonThree**

**}**

{/\* Change code above this line \*/}

**</div>**

**); }**

**}**

# **Render Conditionally from Props**

So far, you've seen how to use if/else, &&, and the ternary operator (condition ? expressionIfTrue : expressionIfFalse) to make conditional decisions about what to render and when. However, there's one important topic left to discuss that lets you combine any or all of these concepts with another powerful React feature: props. Using props to conditionally render code is very common with React developers — that is, they use the value of a given prop to automatically make decisions about what to render.

In this challenge, you'll set up a child component to make rendering decisions based on props. You'll also use the ternary operator, but you can see how several of the other concepts that were covered in the last few challenges might be just as useful in this context.

*The code editor has two components that are partially defined for you: a parent called GameOfChance, and a child called Results. They are used to create a simple game where the user presses a button to see if they win or lose.*

*First, you'll need a simple expression that randomly returns a different value every time it is run. You can use Math.random(). This method returns a value between 0 (inclusive) and 1 (exclusive) each time it is called. So for 50/50 odds, use Math.random() >= .5 in your expression. Statistically speaking, this expression will return true 50% of the time, and false the other 50%. In the render method, replace null with the above expression to complete the variable declaration.*

*Now you have an expression that you can use to make a randomized decision in the code. Next you need to implement this. Render the Results component as a child of GameOfChance, and pass in expression as a prop called fiftyFifty. In the Results component, write a ternary expression to render the h1 element with the text You Win! or You Lose! based on the fiftyFifty prop that's being passed in from GameOfChance. Finally, make sure the handleClick() method is correctly counting each turn so the user knows how many times they've played. This also serves to let the user know the component has actually updated in case they win or lose twice in a row.*

**class Results extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return (**

**<h1>**

**{this.props.fiftyFifty ? "You Win!" : "You Lose!"}**

**</h1>**

**)**

**};**

**};**

**class GameOfChance extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**counter: 1**

**}**

**this.handleClick = this.handleClick.bind(this);**

**}**

**handleClick() {**

**this.setState({**

**counter: this.state.counter + 1 // change code here**

**});**

**}**

**render() {**

**const expression = Math.random() >= 0.5 ? true : false**

**return (**

**<div>**

**<button onClick={this.handleClick}>Play Again</button>**

{ /\* change code below this line \*/ }

**<Results fiftyFifty={expression} />**

{ /\* change code above this line \*/ }

**<p>{'Turn: ' + this.state.counter}</p>**

**</div>**

**);**

**}**

**};**

# **Change Inline CSS Conditionally Based on Component State**

At this point, you've seen several applications of conditional rendering and the use of inline styles. Here's one more example that combines both of these topics. You can also render CSS conditionally based on the state of a React component. To do this, you check for a condition, and if that condition is met, you modify the styles object that's assigned to the JSX elements in the render method.

This paradigm is important to understand because it is a dramatic shift from the more traditional approach of applying styles by modifying DOM elements directly (which is very common with jQuery, for example). In that approach, you must keep track of when elements change and also handle the actual manipulation directly. It can become difficult to keep track of changes, potentially making your UI unpredictable. When you set a style object based on a condition, you describe how the UI should look as a function of the application's state. There is a clear flow of information that only moves in one direction. This is the preferred method when writing applications with React.

*The code editor has a simple controlled input component with a styled border. You want to style this border red if the user types more than 15 characters of text in the input box. Add a condition to check for this and, if the condition is valid, set the input border style to 3px solid red. You can try it out by entering text in the input.*

**class GateKeeper extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: ' '**

**};**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleChange(event) {**

**this.setState({ input: event.target.value })**

**}**

**render() {**

**let inputStyle = {**

**border: '1px solid black'**

**};**

// change code below this line

**if (this.state.input.length > 15) {**

**inputStyle.border = '3px solid red';**

**}**

// change code above this line

**return (**

**<div>**

**<h3>Don't Type Too Much:</h3>**

**<input**

**type="text"**

**style={inputStyle}**

**value={this.state.input}**

**onChange={this.handleChange} />**

**</div>**

**);**

**}**

**};**

# **Use Array.map() to Dynamically Render Elements**

Conditional rendering is useful, but you may need your components to render an unknown number of elements. Often in reactive programming, a programmer has no way to know what the state of an application is until runtime, because so much depends on a user's interaction with that program. Programmers need to write their code to correctly handle that unknown state ahead of time. Using Array.map() in React illustrates this concept.

For example, you create a simple "To Do List" app. As the programmer, you have no way of knowing how many items a user might have on their list. You need to set up your component to dynamically render the correct number of list elements long before someone using the program decides that today is laundry day.

*The code editor has most of the MyToDoList component set up. Some of this code should look familiar if you completed the controlled form challenge. You'll notice a textarea and a button, along with a couple of methods that track their states, but nothing is rendered to the page yet.*

*Inside the constructor, create a this.state object and define two states: userInput should be initialized as an empty string, and toDoList should be initialized as an empty array. Next, delete the null value in the render() method next to the items variable. In its place, map over the toDoList array stored in the component's internal state and dynamically render a li for each item. Try entering the string eat, code, sleep, repeat into the textarea, then click the button and see what happens.*

***Note:****You may know that all sibling child elements created by a mapping operation like this do need to be supplied with a unique key attribute. Don't worry, this is the topic of the next challenge.*

**const textAreaStyles = {**

**width: 235,**

**margin: 5**

**};**

**class MyToDoList extends React.Component {**

**constructor(props) {**

**super(props);**

**// change code below this line**

**this.state = {**

**userInput: ' ',**

**toDoList: [ ]**

**}**

**// change code above this line**

**this.handleSubmit = this.handleSubmit.bind(this);**

**this.handleChange = this.handleChange.bind(this);**

**}**

**handleSubmit() {**

**const itemsArray = this.state.userInput.split(',');**

**this.setState({**

**toDoList: itemsArray**

**});**

**}**

**handleChange(e) {**

**this.setState({**

**userInput: e.target.value**

**});**

**}**

**render() {**

**const items = this.state.toDoList.map(i => <li>{i}</li>); // change code here**

**return (**

**<div>**

**<textarea**

**onChange={this.handleChange}**

**value={this.state.userInput}**

**style={textAreaStyles}**

**placeholder="Separate Items With Commas" /><br />**

**<button onClick={this.handleSubmit}>Create List</button>**

**<h1>My "To Do" List:</h1>**

**<ul>**

**{items}**

**</ul>**

**</div>**

**);**

**}**

**};**

# **Give Sibling Elements a Unique Key Attribute**

The last challenge showed how the map method is used to dynamically render a number of elements based on user input. However, there was an important piece missing from that example. When you create an array of elements, each one needs a key attribute set to a unique value. React uses these keys to keep track of which items are added, changed, or removed. This helps make the re-rendering process more efficient when the list is modified in any way.

**Note:** Keys only need to be unique between sibling elements, they don't need to be globally unique in your application.

*The code editor has an array with some front end frameworks and a stateless functional component named Frameworks(). Frameworks() needs to map the array to an unordered list, much like in the last challenge. Finish writing the map callback to return an li element for each framework in the frontEndFrameworks array. This time, make sure to give each li a key attribute, set to a unique value. The li elements should also contain text from frontEndFrameworks.*

*Normally, you want to make the key something that uniquely identifies the element being rendered. As a last resort the array index may be used, but typically you should try to use a unique identification.*

**const frontEndFrameworks = [**

**'React',**

**'Angular',**

**'Ember',**

**'Knockout',**

**'Backbone',**

**'Vue'**

**];**

**function Frameworks() {**

**const renderFrameworks = frontEndFrameworks.map((item) =>**

**<li key={item}>{item}</li>**

**);** // Change this line

**return (**

**<div>**

**<h1>Popular Front End JavaScript Frameworks</h1>**

**<ul>**

**{renderFrameworks}**

**</ul>**

**</div>**

**);**

**};**

# **Use Array.filter() to Dynamically Filter an Array**

The map array method is a powerful tool that you will use often when working with React. Another method related to map is filter, which filters the contents of an array based on a condition, then returns a new array. For example, if you have an array of users that all have a property online which can be set to true or false, you can filter only those users that are online by writing:

let onlineUsers = users.filter(user => user.online);

*In the code editor, MyComponent's state is initialized with an array of users. Some users are online and some aren't. Filter the array so you see only the users who are online. To do this, first use filter to return a new array containing only the users whose online property is true. Then, in the renderOnline variable, map over the filtered array, and return a li element for each user that contains the text of their username. Be sure to include a unique key as well, like in the last challenges.*

**class MyComponent extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**users: [**

**{**

**username: 'Jeff',**

**online: true**

**},**

**{**

**username: 'Alan',**

**online: false**

**},**

**{**

**username: 'Mary',**

**online: true**

**},**

**{**

**username: 'Jim',**

**online: false**

**},**

**{**

**username: 'Sara',**

**online: true**

**},**

**{**

**username: 'Laura',**

**online: true**

**}**

**]**

**}**

**}**

**render() {**

**const usersOnline = this.state.users.filter(user => user.online === true);** // change code here

**const renderOnline = usersOnline.map(user => <li key={user.username}>{user.username}</li>);**

// change code above

**return (**

**<div>**

**<h1>Current Online Users:</h1>**

**<ul>**

**{renderOnline}**

**</ul>**

**</div>**

**);**

**}**

**};**

# **Render React on the Server with renderToString**

So far, you have been rendering React components on the client. Normally, this is what you will always do. However, there are some use cases where it makes sense to render a React component on the server. Since React is a JavaScript view library and you can run JavaScript on the server with Node, this is possible. In fact, React provides a renderToString() method you can use for this purpose.

There are two key reasons why rendering on the server may be used in a real world app. First, without doing this, your React apps would consist of a relatively empty HTML file and a large bundle of JavaScript when it's initially loaded to the browser. This may not be ideal for search engines that are trying to index the content of your pages so people can find you. If you render the initial HTML markup on the server and send this to the client, the initial page load contains all of the page's markup which can be crawled by search engines. Second, this creates a faster initial page load experience because the rendered HTML is smaller than the JavaScript code of the entire app. React will still be able to recognize your app and manage it after the initial load.

*The renderToString() method is provided on ReactDOMServer, which is available here as a global object. The method takes one argument which is a React element. Use this to render App to a string.*

**class App extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <div/>**

**}**

**};**

// Change code below this line

**ReactDOMServer.renderToString(<App/>)**